AmgT: Algebraic Multigrid Solver on Tensor Cores

Yuechen Lu*, Lijie Zeng*, Tengcheng Wang*, Xu Fu*, Wenxuan Li*, Helin Cheng*, Dechuang Yang*, Zhou Jin*, Marc Casas†, Weifeng Liu*
*Super Scientific Software Laboratory, Dept. of CST, China University of Petroleum-Beijing, China
†Barcelona Supercomputing Center, Spain
{yuechenlu, lijie.zeng, tengcheng.wang, xu.fu, wenxuan.li, helin.cheng, dechuang.yang}@student.cup.edu.cn
jinzhou@cup.edu.cn, marc.casas@bsc.es, weifeng.liu@cup.edu.cn

***Abstract*—Algebraic multigrid (AMG) methods are particularly efficient to solve a wide range of sparse linear systems, due to their good flexibility and adaptability. Even though modern parallel devices, such as GPUs, brought massive parallelism to AMG, the latest major hardware features, i.e., tensor core units and their low precision compute power, have not been exploited to accelerate AMG.**

This paper proposes AmgT, a new AMG solver that utilizes the tensor core and mixed precision ability of the latest GPUs during multiple phases of the AMG algorithm. Considering that the sparse general matrix-matrix multiplication (SpGEMM) and sparse matrix-vector multiplication (SpMV) are extensively used in the setup and solve phases, respectively, we propose a novel method based on a new unified sparse storage format that leverages tensor cores and their variable precision. Our method improves both the performance of GPU kernels, and also reduces the cost of format conversion in the whole data flow of AMG. To better utilize the algorithm components in existing libraries, the data format and compute kernels of the AmgT solver are incorporated into the HYPRE library. The experimental results on NVIDIA A100, H100 and AMD MI210 GPUs show that our AmgT outperforms the original GPU version of HYPRE by a factor of on geomean 1.46×, 1.32× and 2.24×(up to 2.10×, 2.06× and 3.67×), respectively.

***Index Terms*—AMG, SpGEMM, SpMV, tensor core unit, mixed precision**

I. Introduction

The MultiGrid (MG) method is one of the most efficient techniques for solving linear systems of equations. One of the MG variants, the Algebraic MultiGrid (AMG) method, constructs grids directly from the coefficient matrix of a linear system, regardless of whether it has a clear geometric background [1]–[3]. The algebraic approach of AMG makes it possible to efficiently solve systems of equations arising for a wide variety of application domains, such as molecular dynamics simulation [4], electromagnetic field analysis [5], and global weather prediction [6].

Because of its importance, optimizing AMG on modern parallel processors is receiving much attention [7]–[21], and these optimizations are being adopted by high performance numerical libraries such as HYPRE [22], CUSP [23], AmgX [15], JXPAMG [24] and Ginkgo [25]. Among the different approaches, GPU-based acceleration of AMG has significant advantages in terms of graph matching and coloring [15], fine-grained parallelism [16], heterogeneous execution policies and interfaces [17], hybrid sparse format [18], multi-GPU communication [19], as well as mixed precision [20].

However, despite the success of the existing work on GPUs, two major features of modern GPUs, i.e., tensor core units and their ability on low precision computations, have not been exploited for accelerating AMG. Tensor core units compute small dense general matrix-matrix multiplication (GEMM), and have been added for the first time to recent generations of NVIDIA GPUs [26]–[28] and now appear on CPUs and GPUs from more vendors [29]–[31]. Meanwhile, calling tensor cores is typically the only way to unleash the rich power of low precision computations on GPUs. As an example, the tensor core units of the latest NVIDIA H100 GPU deliver 2× peak performance for FP64 (51.2 vs. 25.6, in TFlops) and 7× for FP16 (756 vs. 102.4, in TFlops) over the CUDA cores [28].

We in this work focus on utilizing the tensor cores with variable precision for optimizing the whole procedure of AMG. Specifically, the time-consuming sparse general matrix-matrix multiplication (SpGEMM) in the setup phase, and sparse matrix-vector multiplication (SpMV) in the solve phase, are the major kernels to accelerate. Although there has been much individual research on optimizing SpGEMM [32]–[51] and SpMV [52]–[68], it is still difficult to optimize both kernels in the whole AMG scenario on tensor cores, because of three challenges: (1) storage-wise, how to avoid generating individual matrix instances with different formats for both SpGEMM and SpMV; (2) compute-wise, how to design efficient methods for matching general sparse structures with the strict dense GEMM patterns of tensor cores; (3) precision-wise, how to integrate SpGEMM and SpMV with variable precision into the complete data flow of AMG.

To address the three challenges, we first design a unified sparse format called mBSR, which is a variant of the classic block sparse row (BSR) format. The mBSR format stores a sparse matrix in a group of dense tiles of size 4-by-4 and uses a bitmap to present the positions of nonzeros in each tile. Then, on top of this data format, we propose AmgT, a new AMG solver that utilizes the tensor core and mixed precision ability of the latest GPUs during multiple phases of the AMG algorithm. AmgT is based on new SpGEMM and SpMV algorithms capable of accelerating computation using both tensor cores and CUDA cores, depending on the sparsity of a tile. Our SpGEMM algorithm analyzes the matrix data and groups all block rows into eight bins, performs a two-step hash-style symbolic computation to obtain the position information of tiles in the resulting matrix, and uses

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

a combination of tensor cores and CUDA cores for numeric computation. Our SpMV algorithm uses load-balanced and adaptive selection of computation cores strategies, and implements the use of a hybrid of tensor cores and CUDA cores to improve overall performance. Finally, the new SpGEMM and SpMV algorithms that use high and low precisions are called in fine and coarse grids, respectively, to efficiently leverage the tensor core computing capacity.

We implement tensor core optimized SpGEMM and SpMV kernels with the unified sparse format, and incorporate them into the HYPRE library [22] to maximize their use along with many other components. Two NVIDIA GPUs A100 (Ampere) and H100 (Hopper) and one AMD GPU MI210 (CDNA2) are used as our platform, and 16 representative sparse matrices of various kinds from the SuiteSparse Matrix Collection [69] are evaluated.

The experimental results show that our double-precision AmgT is on geomean 1.46×, 1.32× and 2.24× (up to 2.10×, 2.06× and 3.67×) faster than the original GPU version of HYPRE v2.31.0. On A100 and H100, our mixed-precision AmgT is further on geomean 1.03× and 1.04× (up to 1.08× and 1.14×) faster than our double-precision AmgT. For the performance on multi-GPU (eight A100 cards), our double-precision AmgT is on geomean 1.35× (up to 1.84×) than HYPRE, while our mixed-precision AmgT is on geomean 1.06× (up to 1.11×) faster than double-precision AmgT. Also, the standalone kernel tests show that our SpGEMM is faster than cuSPARSE and rocSPARSE SpGEMM by a factor of geomean 3.09×, 2.40× and 4.67× (up to 7.61×, 6.11× and 5.96×), and SpMV outperforms cuSPARSE and rocSPARSE SpMV by a factor of geomean 1.34×, 1.19× and 2.92× (up to 2.21×, 2.09× and 6.70×) on the three GPUs, respectively.

This work makes the following contributions:

- We design a unified sparse matrix format that supports both SpGEMM and SpMV;
- We propose tensor core and mixed precision friendly SpGEMM and SpMV kernels;
- We develop the AmgT solver and incorporate it into the HYPRE library;
- We show significant performance gain of AmgT on two latest NVIDIA GPUs and one AMD GPU.

II. Key Components of AMG

The AMG method mainly consists of two phases: setup and solve, shown in Algorithms 1 and 2, respectively. The setup phase constructs a multi-level hierarchy of grids, while the solve phase iterates to solve the linear system in the hierarchical structure.

A. Setup Phase

The setup phase is an iterative process that produces a hierarchy of grids, represented by sparse matrices. Each iteration of the setup phase includes three major steps: (1) coarsening (line 3 in Algorithm 1) partitions nodes of grid Ω^k, which is represented by the sparse matrix A^k, into fine and coarse point sets F^k and C^k by identifying strong and weak connections within the grid. Set C^k becomes the next grid, Ω^{k+1}; (2) interpolation (line 4) requires one instance of the SpGEMM kernel [35] to generate interpolation and restriction operators P^k and R^K for information transfer between fine and coarse grids; (3) Galerkin product (line 5) computes the product $R^kA^kP^k$ via a couple of SpGEMM operations to construct the coarse grid matrix A^{k+1}.

Algorithm 1 An M-level setup phase

Input: Sparse matrix A, Grid Ω ($A^1 = A, \Omega^1 = \Omega$)
Output: A^k, R^k, P^k.
1: $k \leftarrow 1$
2: **while** Ω^k is not small enough **do**
3: Coarsening: Grid Ω^k is partitioned into sets C^k and F^k, and $\Omega^{k+1} \leftarrow C^k$.
4: Interpolation: P^k and $R^k = (P^k)^T$ through one SpGEMM call [35].
5: Galerkin product: $A^{k+1} = R^kA^kP^k$ through two SpGEMM calls.
6: **end while**
7: $M \leftarrow k$

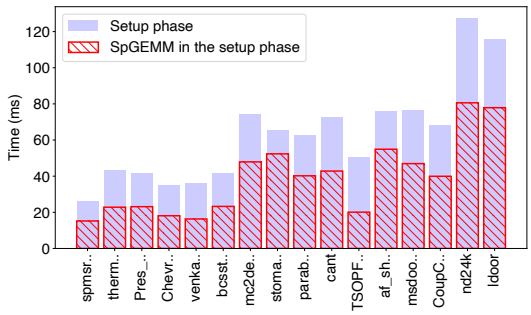

Fig. 1. Execution time breakdown of the setup phase on an H100 GPU.

We consider 16 representative matrices to profile the AMG setup phase (details of the matrices are provided in Table II). Figure 1 displays the contribution to the total setup time of the SpGEMM calls. As can be seen, the three calls (one in the interpolation step and two in the Galerkin product step) take on average 59.22% of the execution time of the setup phase. Therefore, SpGEMM can be the fundamental routine to optimize for accelerating the setup phase.

Algorithm 2 A solve phase within V-cycle

Input: A^k, R^k, P^k, x^k, b^k.
Output: Solution vector x^k.
1: **for** $k = 1, 2, ..., M-1$ **do**
2: Pre-smoothing: $x_{i+1}^k = x_i^k + D^{-1}(b^k - A^k x_i^k)$ μ_1 times through μ_1 SpMV calls.
3: Residual: $r^k = b^k - A^k x^k$ through one SpMV call.
4: Restriction: $b^{k+1} = R^k r^k$ through one SpMV call.
5: **end for**
6: Coarsest level solving: $A^M x^M = b^M$ through an iterative or direct method.
7: **for** $k = M-1, M-2, ..., 1$ **do**
8: Interpolation: $e^k = P^k x^{k+1}$ through one SpMV call.
9: Correction: $x^k \leftarrow x^k + e^k$.
10: Post-smoothing: $x_{i+1}^k = x_i^k + D^{-1}(b^k - A^k x_i^k)$ μ_2 times through μ_2 SpMV calls.
11: **end for**

B. Solve Phase

The solve phase includes four major steps: (1) smoothing (lines 2 and 10 in Algorithm 2) reduces high-frequency components in the error through multiple Jacobi iterations, which requires μ_1 SpMV calls in pre-smoothing phase and μ_2 SpMV calls in post-smoothing; (2) residual and restriction

(lines 3 and 4) compute smoothing error to the coarse grids through two SpMV calls; (3) the coarsest level solving (line 6) gives a solution of the system at the coarsest level by using an iterative or direct method like PanguLU [70]; (4) interpolation and correction (lines 8 and 9) make smoothing error back to the fine grids through one SpMV call. Finally, the iteration steps are repeated until the relative residual error satisfies the convergence criteria.

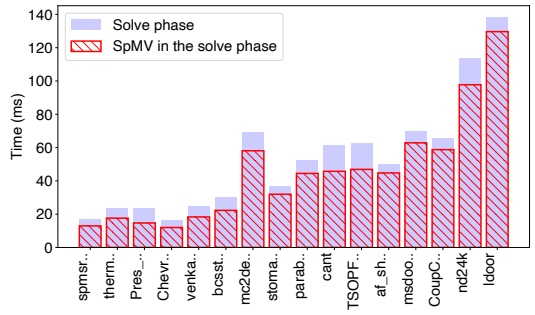

Fig. 2. Execution time breakdown of the solve phase on an H100 GPU.

Figure 2 shows a breakdown of the solve execution time for the 16 matrices we considered. On average, 80.23% of the total solve time is spent on running the SpMV kernel, making it the most critical routine to optimize. Moreover, to accelerate the solve phase, preconditioners such as the preconditioned conjugate gradient (PCG) can also be used for faster convergence. The preconditioners often include a number of SpMV calls, making SpMV more time-consuming in the solve phase.

III. Tensor Core Units and Mixed Precision for AMG Acceleration

The tensor core units of recent GPUs offer specific hardware for small GEMM of fixed sizes, and typically deliver a high operations per second throughput for low precision arithmetic. Taking a double precision 8-by-8-by-4 tensor core unit as an example, it in four cycles multiples A of size 8-by-4 and B of size 4-by-8, and gives C of size 8-by-8. In addition, the peak performance difference of multiple floating point formats can be large. For example, on both A100 and H100, the peak performance for low precision formats delivers larger advantages over CUDA cores (*e.g.*, 7× in FP16) than high precision formats (2× in FP64).

Despite the advantages of multiple precision compute power from the tensor cores, exploiting them for the sparse matrix computations involved in AMG is not straightforward. Challenges arise from three distinct aspects:

(1) *Unified sparse format:* Much research has been done on individual optimizations for SpGEMM [32]–[51] and SpMV [52]–[68], and these techniques typically require new sparse matrix formats. But these formats are not directly applicable to AMG since this method generates new sparse matrices in its multi-level grids, and frequent matrix format conversions can incur significant overhead and lead to low overall performance. Therefore, designing a unified sparse format for both SpGEMM and SpMV becomes a key challenge in our AmgT.

(2) *Tensor core friendly sparse kernels:* Tensor cores are designed to accelerate dense GEMM and have strict size constraints for input and output matrices. The nonzeros of sparse matrices involved in AMG, however, are distributed irregularly and entail non-sequential memory access patterns. This incongruity with the tensor cores complicates their direct exploitation in the context of AMG. To resolve this disparity, it is essential to develop new SpGEMM and SpMV algorithms able to map the nonzeros stored in the unified sparse format onto the regular patterns required by the tensor cores.

(3) *Mixed precision in the AMG data flow:* Tensor cores provide superior performance in low precision, and recent research [20] has indicated that AMG can be accelerated using low precision kernels in the coarse layers without affecting the final convergence. Thus, to take more advantage of the tensor cores, we need to design multiple precision SpGEMM and SpMV kernels to be used on the best suitable layers of AmgT. In addition, integrating AmgT into existing libraries, e.g., HYPRE [22], makes it possible to take advantage of some already existing and high-quality software components.

IV. The AmgT Solver

A. Overview

This section introduces the AmgT solver. AmgT is composed of several components: (1) a unified data structure called mBSR for storing the matrices processed in AMG (Section IV.B); (2) efficient algorithms for SpGEMM (Section IV.C) and SpMV (Section IV.D) for utilizing the computational power of both tensor cores and CUDA cores; (3) a data flow for format conversion and kernel calls, and for using multiple precisions in the fine and coarse layers (Section IV.E).

B. Unified Sparse Matrix Format

We develop a unified sparse matrix data structure for both SpGEMM and SpMV operations, called the mBSR format, which is a variant of the classic BSR format. Figure 3 shows an example of a sparse matrix of size 8-by-8, divided into three blocks. The mBSR format is composed of two levels: the first level stores the position information of blocks using two arrays: the `blcPtr` array of size blc_row+1, where blc_row is the number of block rows of the matrix, that stores the memory offsets of the first blocks in block-rows, and the `blcIdx` array of size blc_num, where blc_num is the number of blocks in the matrix, that stores the column index of each block. The second level contains two arrays: the `blcVal` array of size $blc\_num \times 4 \times 4$, that stores the values of elements within each block, and the `blcMap` array of size blc_num that stores bitmap indicating the existence of nonzeros of each block. The four arrays are plotted in Figure 3.

Considering that the dimension sizes of both input and output matrices supported by the tensor core are multiples of four, we set the block size to 4-by-4, so that the desired sizes of a tensor core can be pieced together by simple operations.

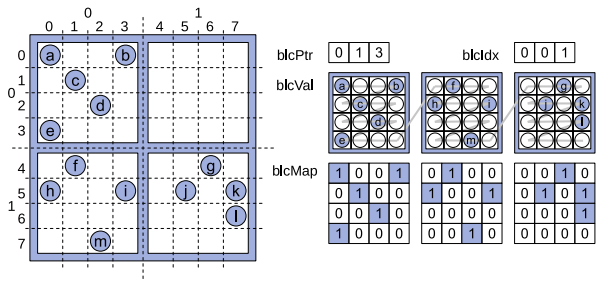

Fig. 3. An example matrix of 8-by-8 uses a two-level structure to store its nonzeros: the first level stores the position information of blocks in arrays `blcIdx` and `blcPtr`; the second level stores the values and the positions of nonzeros in arrays `blcVal` and `blcMap`, respectively.

Unlike the classic BSR format, we use the bitmap that stores the positions of nonzeros within each block, and the 16-bit message in a bitmap for each block is stored by using exactly one unsigned short. This bitmap information makes it possible to efficiently determine the sparsity of each block by binary manipulations, so that we can choose the proper cores to compute.

Algorithm 3 A pseudocode of symbolic SpGEMM

Input: `matA`, `matB` (`BlcPtr`, `BlcCid`, `BlcMap`)
Output: `matC` (`BlcPtr`, `BlcCid`)

```
1:  __shared__ hash_table[SM_SIZE]
2:  for i = BlcPtrA[rowid] to BlcPtrA[rowid + 1] do
3:      mapA ← BlcMapA[i]
4:      cidA ← BlcCidA[i]
5:      for j = BlcPtrB[cidA] to BlcPtrB[cidA + 1] do
6:          mapB ← BlcMapB[j]
7:          mapC ← BITMAPMULTIPLY(mapA, mapB)
8:          if mapC then
9:              Use hash method record nnzC_row          (in step1)
                Use hash method record cidC              (in step2)
10:         end if
11:     end for
12: end for
13: BlcPtrC[rowid] ← nnzC_row                            (in step1)
    Compress hash_table                                  (in step2)
14: Sort hash_table and write back BlcCidC               (in step2)
```

C. SpGEMM for the Setup Phase

The SpGEMM operation, which multiplies a sparse matrix A by another sparse matrix B to yield a sparse matrix C, is a time-consuming operation within the AMG setup phase. In order to utilize tensor cores for accelerating SpGEMM, we use the mBSR format proposed in the last subsection as the unified basis for computing the block-wise nonzeros. In the SpGEMM algorithm accelerated by tensor cores, it is necessary to piece several blocks into a new block with a shape compatible with the tensor cores, prior to invoking the tensor core instruction (matrix multiply-accumulate operation, or mma instruction) to conduct the small GEMM computation. Furthermore, due to the irregular distribution of the blocks in sparse matrices, it is difficult to simultaneously satisfy the shapes supported by the tensor core and the alignment of blocks in the same row and column in the matrices A and B. Therefore, we only use half of the results obtained from the tensor cores to ensure the simplicity of the piecing operation.

Figure 4 represents on the top left-hand side the overall flow of our SpGEMM. The first step of our algorithm is a data analysis process to compute the number of intermediate products per row, which guides a binning operation on the block-rows of C. Secondly, our algorithm performs a two-step symbolic computation with hash tables to compute the number and column indices of blocks in C. Finally, a hybrid method that uses tensor cores and CUDA cores performs a small GEMM to get the values in C and their bitmaps.

1) Data analysis and binning: By accumulating the number of blocks in block-rows of B, which block-rows corresponded to the column indices of all the blocks in one block-row of A, we obtain the number of intermediate product blocks per block-row in C, denoted as *Cub_per_row*. Based on the different quantities, all block-rows of C are grouped into eight different bins. The grouping standard starts from a minimum of 128 and increases by powers of 2 until it reaches 8192. That is, the first bin contains block-rows with *Cub_per_row* less than 128, and the last bin contains block-rows with *Cub_per_row* greater than or equal to 8192.

2) Two-step symbolic computation: We employ a hash approach, and use a thread-block to compute a block-row of C and execute two symbolic computations to determine the positions of each block within the matrix C. For block-rows in different bins, hash tables of variable lengths are allocated in shared memory. In step 1, we utilize the hash method to account for the number of blocks in each block-row of matrix C. Upon completion of this counting, a prefix sum operation is performed to determine the array `BlcPtrC`. By accessing the last value of `BlcPtrC`, the number of blocks can be determined in the entire matrix C, allowing the allocation of memory on the device for arrays `BlcIdxC`, `BlcMapC` and `BlcValC`. Subsequently, the same hash method is executed once again, followed by compression and sorting of the hash table, thereby acquiring the column indices for each block in C. This phase is mainly used for numeric computation using tensor cores and CUDA cores.

3) Numeric Computation: Numeric SpGEMM similarly employs row-wise parallelism, wherein a warp (comprising 32 threads) computes a block-row of C, circumventing conflicts that may arise from multiple threads concurrently accumulating results within one block. Algorithm 4 presents the pseudocode for the numeric SpGEMM. Initially, the row index is determined by the warp index, followed by iterating through each block of A in the specified row, acquiring the bitmap *mapA* for blockA. By using *mapA*, the number of nonzeros within blockA is obtained, which subsequently informs the differentiation of the computation modes for the next action.

When the number of nonzeros in blockA is no less than 10, a warp-level tensor core computation mode is employed. Since the smallest size supported by the tensor core is 8-by-8-by-4, we piece together 8-by-4 blocks as the input $fragA$, 4-by-8 blocks as the input $fragB$, and allocate 8-by-8 space for the output blocks $fragC$ in advance. The entire computation process using tensor cores is represented in the middle plot of Figure 4. The process is divided into the following steps:

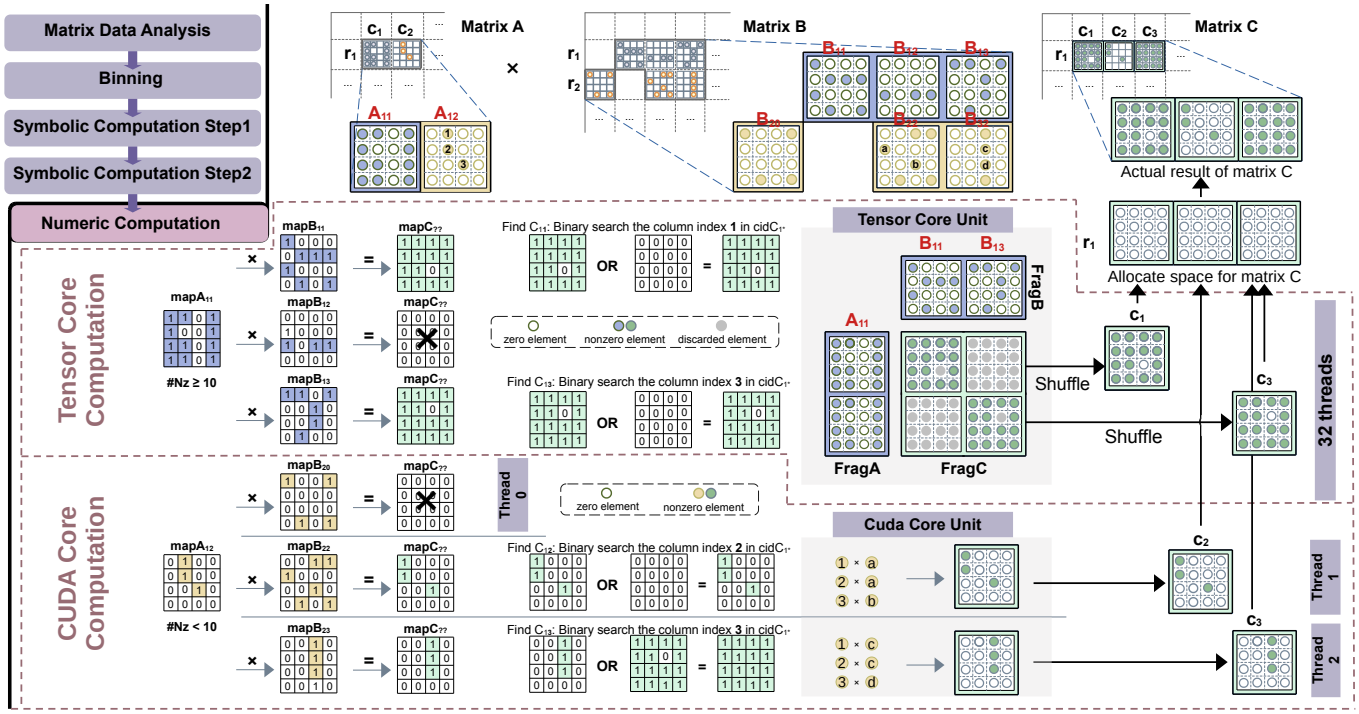

Fig. 4. An example of the numeric computation process of multiplying two sparse matrices, with emphasis on the computation of one block-row of the resulting matrix. In our SpGEMM algorithm flow: the step of matrix data analysis computes the number of intermediate blocks in each block-row of C; the step of binning divides all block-rows into eight bins based on the quantity obtained in the previous step; the two-step symbolic computation gets the positions of all blocks in the matrix C; and the final numeric computation uses the hybrid of tensor cores and CUDA cores to compute the values of matrix C and obtains the bitmap. In the numeric computation process: first, the number of nonzeros of blockA is obtained by reading the bitmap of blockA, which in turn determines whether to use tensor cores for computation; then, we use bitmap multiplication to find the valid blockB in order to prepare data for tensor core computation; next, the position of blockC in C is found by binary search and the map is written back to it, which is also the location where the values of blockC are written back to; finally, we call tensor cores to perform GEMM to get the values of blockC, use the shuffle instructions to extract the result and write that back to C or call CUDA cores to multiply the nonzeros from blockA and blockB to get the result and write it back to the matrix C.

(1) The first step is to prepare the data for $fragA$. To expedite the assembly of the anticipated shape and conform to the computational logic, we replicate the predetermined blockA into $fragA$. That is, the values of two 4-by-4 blocks in $fragA$ are loaded from the same blockA. This approach allows us to directly select two valid blockBs from the same block-row of the matrix A. Figure 4 illustrates under the Tensor Core Unit tag an example of $fragA$ containing two identical 4-by-4 blocks.

(2) The second step is to prepare the data for $fragB$. Not every blockB multiplied with blockA can produce a result blockC containing nonzeros certainly, so we determine the validity of blockB by performing bitmap multiplication in advance, as shown in lines 7-9 in Algorithm 4 and on the left-hand side of Figure 4 middle plot. After that, we load two consecutive valid blockBs into $fragB$ to complete the data preparation of $fragB$.

(3) The next step is to find $cidB$ in the `BlcCidC` array by using binary search to determine the memory offset of the resulting blockC and write the $mapC$ obtained by bitmap multiplication back into the `BlcMapC` array using the bit operation `OR`. Figure 4 represents this process in the central part of its middle plot.

(4) The final step is to call the mma instruction to perform a small GEMM operation and accumulate the results. The data in $fragA$, $fragB$ and $fragC$ are stored in registers spread across all 32 CUDA threads in a warp, and the mma instruction is co-executed by 32 threads together. We need to extract the values of valid blocks in $fragC$ by using the shuffle instructions and accumulate these values to `BlcValC` array. The right-hand side of Figure 4 middle part illustrates the shuffle operations.

Conversely, when the number of nonzeros in blockA is less than 10, a thread-level computation mode is adopted using CUDA cores instead of tensor cores, that is, the operation of multiplying blockA by blockB to get blockC is executed by one thread. Similarly to the first three steps of tensor core computation, we need to identify the blockB that can produce a resulting matrix containing nonzeros and then use binary search to determine the position where the result is written back to. The thread then locates the positions of nonzeros in blockA and blockB by accessing the bitmaps, computes the corresponding values using CUDA cores, gets the results, and writes them back to the `BlcValC` array. The bottom part of Figure 4 illustrates the numeric SpGEMM with CUDA cores.

Algorithm 4 A pseudocode of numeric SpGEMM

Input: matA, matB, matC (BlcPtr, BlcCid, BlcMap, BlcVal)
Output: matC (BlcMap, BlcVal)

1: **for** i = BlcPtrA[$rowid$] **to** BlcPtrA[$rowid+1$] **do**
2: $mapA \leftarrow$ BlcMapA[i]
3: **if** POPCOUNT($mapA$) ≥ 10 **then**
▷ **Tensor Core Compute, warp-level**
4: $cidA \leftarrow$ BlcCidA[i]
5: get $fragA$ from BlcValA
6: **for** j = BlcPtrB[$cidA$] **to** BlcPtrB[$cidA+1$] **do**
7: $mapB \leftarrow$ BlcMapB[j, $j+1$]
8: $mapC \leftarrow$ BITMAPMULTIPLY($mapA$, $mapB$)
9: **if** $mapC$ **then**
10: get $fragB$ from BlcValB
11: Binary Search $cidB$ from BlcCidC
12: Accumulate $mapC$ to BlcMapC
13: MMA_884($fragC$, $fragA$, $fragB$)
14: Extract result from $fragC$ by shuffle instructions
15: Accumulate result to BlcValC
16: **end if**
17: **end for**
18: **else**
▷ **CUDA Core Compute, thread-level**
19: $cidA \leftarrow$ BlcCidA[i]
20: **for** j = BlcPtrB[$cidA$] **to** BlcPtrB[$cidA+1$] **do**
21: $mapB \leftarrow$ BlcMapB[j, $j+1$]
22: $mapC \leftarrow$ BITMAPMULTIPLY($mapA$, $mapB$)
23: **if** $mapC$ **then**
24: Binary Search BlcCidB from BlcCidC
25: Loop through each bit in $MapC$, compute and get $valC$
26: Accumulate $mapC$ to BlcMapC
27: Accumulate $valC$ to BlcValC
28: **end if**
29: **end for**
30: **end if**
31: **end for**

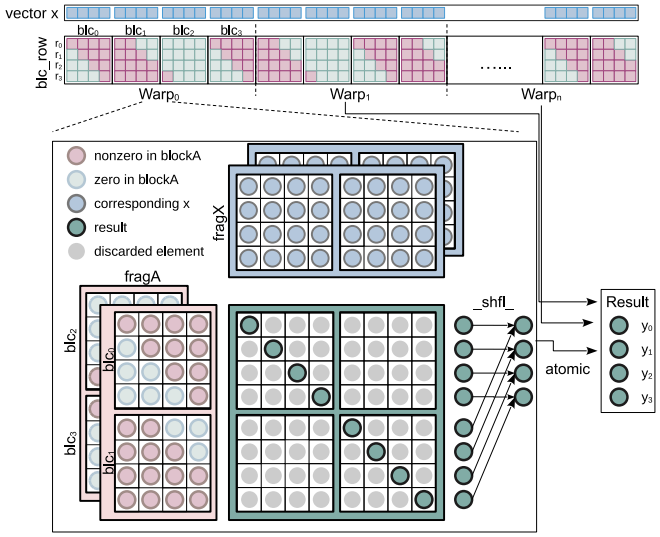

Fig. 5. An example of our load-balanced SpMV algorithm that utilizes tensor cores for computation. In this SpMV algorithm, each warp is allocated the same number of blocks to ensure load balancing. The tensor core instruction is called multiple times within a warp to perform GEMM computation and the results are accumulated to individual fragments. Lastly, the results are extracted by the shuffle instructions and written back to vector y.

D. SpMV for the Solve Phase

The SpMV algorithm performs the multiplication of a sparse matrix A by a dense vector x to obtain a dense vector y, and is a frequently called operation in the solve phase. In our SpMV algorithm, we develop strategies for adaptive selecting load balancing methods and computation cores, and implement the use of a hybrid of tensor cores and CUDA cores.

1) Adaptive Selection: We get a parameter *variation* of the matrix through data preprocessing to evaluate the balance of the distribution of blocks. The value of this parameter makes it possible to determine whether to invoke the load balanced strategy to execute the computation. Regarding load balancing, we fix the workload of each warp to 64 blocks. A number of warps collaboratively complete the computation of one block-row in the matrix. The choice between CUDA or tensor core is made based on the parameter representing the average number of nonzeros within a block, denoted as *avg_nnz_blc*. When *avg_nnz_blc* is greater than or equal to 10, the kernel uses tensor cores for computation. Conversely, the kernel employing CUDA cores is utilized.

2) Tensor Core Computation: Due to the shape restriction on the input matrix shape to a minimum of 8-by-4, we should compute two blocks per call to use the tensor core. As shown in Figure 5, firstly, we need to determine the offset of the first block to be computed by the current warp, then load the values of the two continuous blocks into the register $fragA$ of the 32 threads in this warp. Following this, the values of the vector x corresponding to these two block positions are mapped onto the register $fragB$. After the data preparation, the mma instruction of the tensor cores is called to perform a small dense GEMM. Ultimately, the results of all these blocks are accumulated in $fragY$, and we extract the values on the diagonal of $fragY$ to write back to the vector y.

3) CUDA Core Computation: For matrices that are particularly sparse, the efficiency of using tensor cores often diminishes. Therefore, we opt for the utilization of CUDA cores to carry out the computations. We set the computation that a block is processed collectively by four threads, with each thread being responsible for the nonzeros of one row within the block. As shown in lines 10-11 of Algorithm 5, each thread determines the positions of nonzeros by the bitmap corresponding to its block, multiples these elements by the values of the vector x, and stores the results in the register *res*. Subsequently, a warp-level sum operation is performed, resulting in the writing back of the results to the vector y.

E. Full Data Flow

Figure 6 shows the full data flow of AmgT, where the solid-line arrows represent operations, and the dashed-line arrows represent data transfer. Although the AmgT SpGEMM and SpMV routines use the mBSR format to complete the most time-consuming AMG phases, there are still several components, e.g., coarsening and solving the coarsest level, requiring the CSR format. During the setup phase, AmgT performs CSR2MBSR format conversion on matrices that require SpGEMM operations. In addition, AmgT performs MBSR2CSR on the matrix resulted from the *RAP* operation.

In the setup phase, the input matrix A^1 is initially read and stored in the CSR format, followed by the coarsening operation, as Figure 6 indicates ❶. Prior to the interpolation

Algorithm 5 A pseudocode of SpMV using CUDA Cores

Input: `BlcPtrA`, `BlcCidA`, `BlcMapA`, `BlcValA`, `vecX`, `ridWarp`, `rptWarp`
Output: `vecY`

```
groupid ← laneid >> 2
tid_in_group ← laneid & 3
rowid ← ridWarp[warpid]
start ← rptWarp[rowid] × WARP_CAPACITY
end ← start + WARP_CAPACITY
for i = start + groupid to end stride 8 do
    mapA ← BlcMapA[i]
    for j = 0 to BSR_N do
        idx ← tid_in_group × BSR_N + j
        if GETBIT(mapA,idx) then
            Compute temporary result of vecY
        end if
    end for
end for
Call WarpLevelSum to get the result
if laneid < 4 then
    Write back the result to vecY
end if
```

operation ❷, the intermediate matrices in the CSR format involved in ❷ invoke a `CSR2MBSR` format conversion ❹ to get the mBSR format. Subsequently, the SpGEMM based on the mBSR format in ❷ is executed. After that, we can directly obtain the matrices R^1 and P^1 in the mBSR format. The matrices A^1, R^1 and P^1 in the mBSR format undergo the Galerkin product, i.e., the *RAP* operation, ❸, resulting in the next-level mBSR-formatted matrix A^2. Next, the matrix A^2 invokes an `MBSR2CSR` format conversion ❺ to generate the matrix A^2 in the CSR format. The aforementioned process is iteratively executed until the coarsest M-th level is reached.

In the solve phase, each level sequentially performs pre-smoothing ❻ and restriction ❼ operations from top to bottom on the mBSR-formatted matrices A^k and R^k obtained in the setup phase. Operations ❻ and ❼ can also be completed by invoking the SpMV operation based on the mBSR format, until reaching the M-th level. At the M-th level, the matrix A^M in the CSR format (after ❺ already completed in the setup phase) is solved by employing an iterative or direct method ❿. Finally, each level again uses the mBSR-formatted matrices A^k and P^k obtained during the setup phase to execute the interpolation ❽ and post-smoothing ❾ operations in a bottom-top manner by SpMV.

Within the data flow of AmgT, we directly use a configuration proposed by Tsai et al. [20], and set the various levels involving SpGEMM and SpMV with three precisions: the first level employs double precision, the second level utilizes float precision, while the remaining levels use half precision. The data precision conversions with very low costs will be completed before calling the kernels.

F. Incorporation into HYPRE

In addition to running as a standalone library, AmgT is also incorporated into the HYPRE library to work with more components. Specifically, the arrays of our mBSR format with a prefix `AmgT_mBSR_` are added to HYPRE's `hypre_CSRMatrix` data structure, then after a format conversion `AmgT_CSR2mBSR`, our SpGEMM and SpMV kernels named `AmgT_mBSR_SpGEMM` and `AmgT_mBSR_SpMV` can be called in the `hypre_CSRMatrixMultiplyDevice` and `hypre_CSRMatrixMatvecDevice2` functions, respectively. In this way, with a minimal interface change, AmgT can directly bring performance gains to the AMG components using the two kernels.

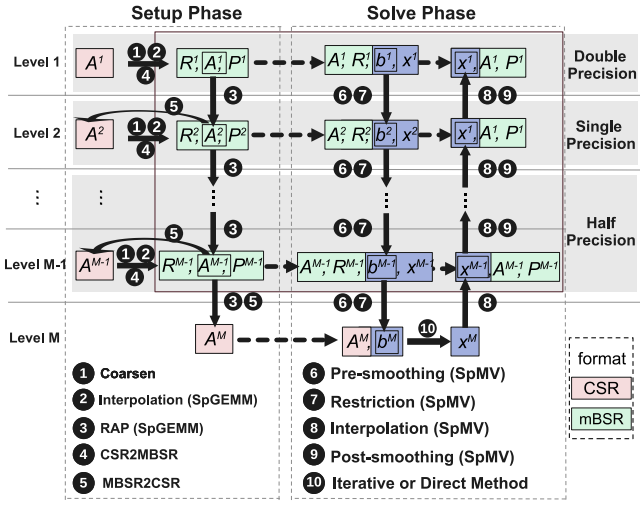

Fig. 6. The full data flow of AmgT exemplified by an M-level grid and iteration of one V-cycle.

V. Evaluation

A. Experimental Setup

We compare our AmgT solver with the original GPU version of the latest HYPRE v2.31.0 calling cuSPARSE v12.2 on two NVIDIA GPUs A100 and H100 and rocSPARSE v6.1.2 on one AMD GPU MI210 (see Table I for specifications), respectively.

TABLE I
The specification of the A100, H100 and MI210 GPUs.

NVIDIA and AMD GPUs	Precision	CUDA/Radeon Core	Tensor/Matrix Core
A100 (Ampere) PCIe	FP64	9.7 TFlops	19.5 TFlops
6912 CUDA cores	FP32/TF32	19.5 TFlops	156 TFlops
80 GB, 1.94 TB/s	FP16	78 TFlops	312 TFlops
H100 (Hopper) SXM5	FP64	33.5 TFlops	66.9 TFlops
16896 CUDA cores	FP32/TF32	66.9 TFlops	494.7 TFlops
64 GB, 2.02 TB/s	FP16	133.8 TFlops	989.4 TFlops
MI210 (CDNA2) PCIe	FP64	22.6 TFlops	45.3 TFlops
6656 Stream Processors	FP32	22.6 TFlops	45.3 TFlops
64 GB, 1.6 TB/s	FP16	181.0 TFlops	181.0 TFlops

To align the comparison, AmgT works inside HYPRE as shown in Section IV.F. For both libraries, we set the same components and parameters, e.g., coarsening (PMIS, $str\_thr = 0.25$, $max\_row\_sum = 0.8$, $max\_coarse\_size = 3$), interpolation (Extended+i with truncation options, $trunc\_fact = 0.1$, $max\_elmts = 4$), smoother (L1_Jacobi, $num\_sweep = 1$). The numbers of calls to SpGEMM (determined by the number of levels) and SpMV (determined by the number of smoothing and iterations) are also identical.

In the setup phase, we make the number of levels no larger than seven, which means that, excluding the coarsest grid, each of the remaining six levels takes one SpGEMM operation in interpolation and two in *RAP*, totaling 18 SpGEMM calls.

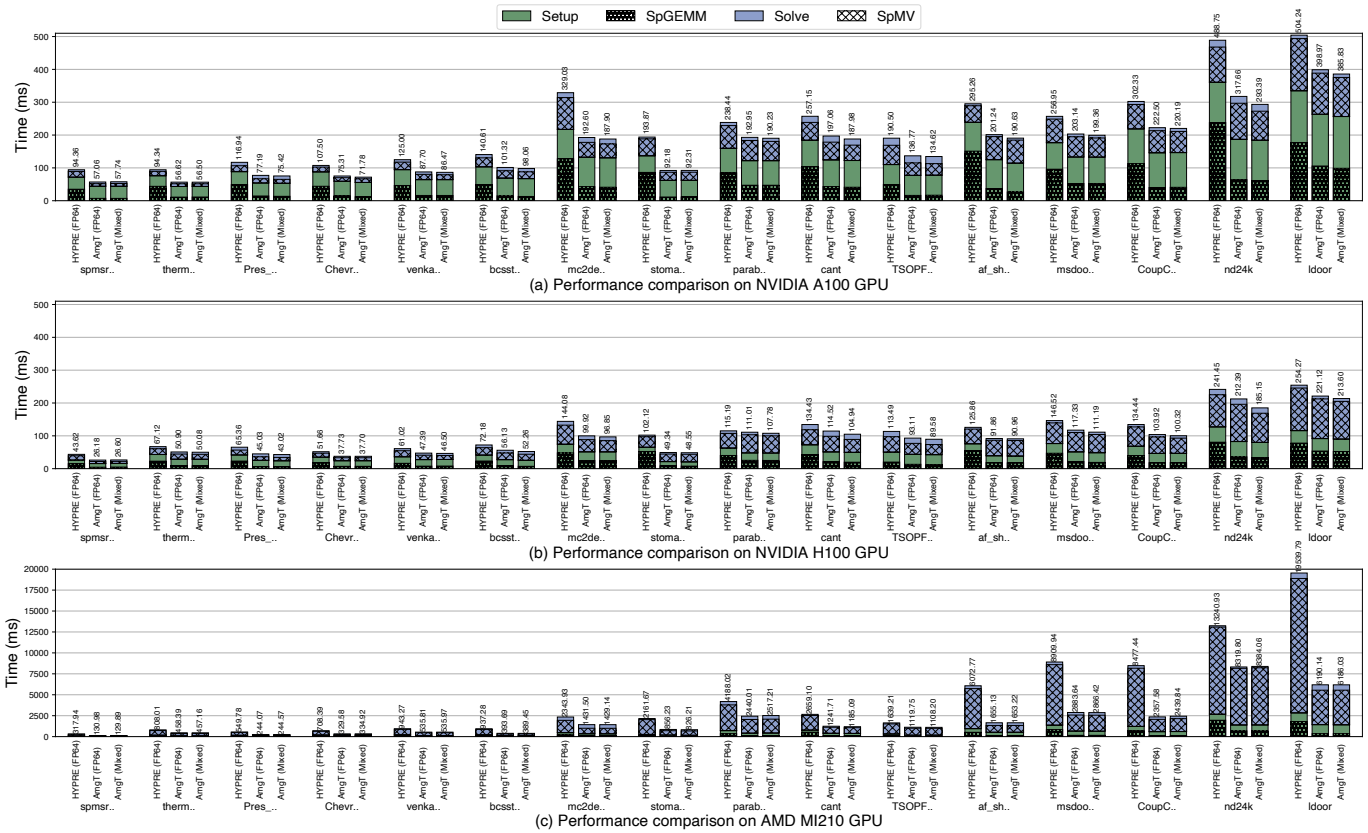

Fig. 7. Performance comparison of the baseline HYPRE using FP64 cuSPARSE and FP64 rocSPARSE kernels, our AmgT using FP64, and our AmgT using mixed-precision running on the 16 representative matrices on two NVIDIA GPUs A100 and H100 and one AMD GPU MI210. In this figure, the x-axis represents the matrices running HYPRE (FP64), AmgT (FP64), and AmgT (Mixed precision), while the y-axis represents the execution time. The total time comprises the setup phase, which includes SpGEMM execution and other operations, and the solve phase, which involves SpMV execution and other operations. The subfigures (a), (b) and (c) represent the performance on an A100, an H100 and an MI210, respectively.

In the solve phase, we set the maximum number of iterations to 50, regardless of convergence. If the coarsest level uses a direct method, in a single V-cycle, each level calls SpMV five times (during the five phases in Algorithm 2). Thus, for a 7-level grid, excluding the coarsest level, the remaining six levels collectively call SpMV 30 times. Additionally, one SpMV runs after each V-cycle to compute the residual. So, one iteration involves 31 SpMV calls. For the solve phase of 50 iterations, a total of 1550 SpMV calls are made. Plus an additional SpMV computing the initial residual, the entire solve phase requires 1551 SpMV calls. If the coarsest level uses an iterative method, each iteration involves 1 or 3 extra SpMVs, resulting in 50 or 150 calls for 50 iterations. Thus, the entire solve phase includes 1601 or 1701 SpMV calls.

Table II lists 16 representative matrices from the SuiteSparse Matrix Collection [69]. The matrices are from diverse groups and are generated from a variety of scientific problems such as thermal, computational fluid dynamics, structural, and power networks. The orders of the matrices range from tens to hundreds of thousands, and the numbers of nonzeros are in the range of hundreds of thousands and tens of millions. Although the dataset could reflect the adaptability of AMG, we do not ensure that all 16 linear systems could converge in the AMG settings mentioned above. To make this work more focused, we mainly demonstrate performance gains from our AmgT.

TABLE II
THE 16 REPRESENTATIVE MATRICES EVALUATED.

Group	Matrix	#Orders	#Nonzeros	#Levels	#SpGEMM	#SpMV
GHS_indef	spmsrtls	29,995	229,947	2	3	351
Schmid	thermal1	82,654	574,458	2	3	351
ACUSIM	Pres_Poisson	14,822	715,804	3	6	551
Chevron	Chevron2	90,249	803,173	2	3	351
Simon	venkat25	62,424	1,717,792	3	6	601
Boeing	bcsstk39	46,772	2,089,294	4	9	851
Williams	mc2depi	525,825	2,100,225	5	12	1101
Norris	stomach	213,360	3,021,648	2	3	351
Wissgott	parabolic_fem	525,825	3,674,625	3	6	601
Williams	cant	62,451	4,007,383	7	18	1701
TSOPF	TSOPF_RS_b300_c3	42,138	4,413,449	7	18	1701
Schenk_AFE	af_shell4	504,855	17,588,875	2	3	351
INPRO	msdoor	415,863	20,240,935	3	6	601
Janna	CoupCons3D	416,800	22,322,336	3	6	601
ND	nd24k	72,000	28,715,634	7	18	1701
GHS_psdef	ldoor	952,203	46,522,475	3	6	601

B. Performance Comparison in Double Precision

Figure 7 presents a comparison of the execution time of the 16 matrices on the A100 (top) and H100 (middle) GPUs using cuSPARSE-support HYPRE, double precision AmgT, and mixed precision AmgT. In this figure, the green portion indicates the time taken during the setup phase, with the shadowed overlay representing the time overhead of SpGEMM.

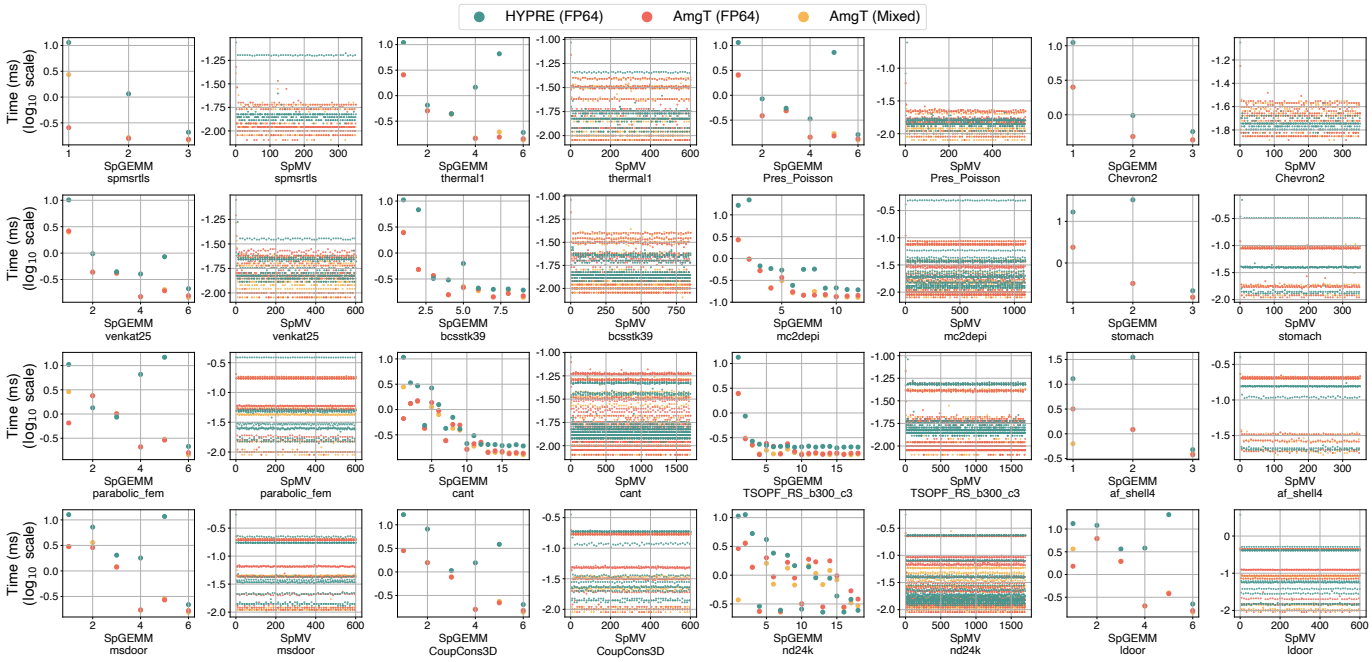

Fig. 8. Performance of every SpGEMM and SpMV run in HYPRE, AmgT, and AmgT of mixed precision on H100. The x-axis shows the time sequence of running the two kernels, the y-axis is the kernel execution time, and the three colored dots represent the three methods. Here, each matrix has two subfigures.

The blue portion denotes the time during the solve phase, and the shadowed overlay on this indicates the time overhead of SpMV. Taking into account the total execution time, our AmgT (FP64), in comparison to HYPRE, yields performance improvements of 1.46× and 1.32× (up to 2.10× and 2.06×) speedups on the A100 and H100 GPUs, respectively.

For the setup phase, compared with HYPRE calling cuSPARSE kernels, AmgT (FP64) achieves a geomean of 1.57× and 1.53× (up to 2.20× and 3.02×) speedups on the A100 and H100 GPUs, respectively, where the execution time of SpGEMM reaches a geomean of 3.09× and 2.40× (up to 7.61× and 6.11×) speedups. Taking the performance of matrix 'cant' as an example, this matrix executed 18 SpGEMM calls in the setup phase, six of which are performed in the interpolation operation and the rest are in the Galerkin product. In this process, our SpGEMM algorithm has overall speedups of 2.38× and 2.02× over the SpGEMM function in cuSPARSE on the two GPUs, respectively; reflecting on the entire setup phase, the execution time of AmgT is 1.48× and 1.42× faster than HYPRE. By utilizing our SpGEMM algorithm, the proportion of SpGEMM execution time in the setup phase is reduced from 58.80% to 41.39%.

With regard to the execution time for the solve phase, our AmgT with FP64 precision, compared to HYPRE that calls cuSPARSE kernels, achieves speedups of 1.24× and 1.13× (up to 1.89× and 1.64×) on the A100 and H100 GPUs, respectively. Further observation of the execution time of SpMV reveals that our SpMV algorithm achieves on average 1.34× and 1.19× (up to 2.21× and 2.09×) speedups compared to the cuSPARSE SpMV method on the two GPUs. Taking the matrix 'venkat25' as an example, it calls SpMV 601 times in the solve phase. From the perspective of the total execution time for these SpMV operations, our mBSR format SpMV algorithm, is 1.43× and 1.37× faster than cuSPARSE on the two GPUs, respectively. Also, when we observe the overall time for the solve phase, our AmgT method with FP64 precision is 1.29× and 1.25× faster than HYPRE. By leveraging the tensor core accelerated SpMV algorithm we provided, the proportion of time devoted to SpMV operations for the matrix 'venkat25' during the solve phase drops from 75.56% to 68.38%.

C. Performance Comparison in Mixed Precision

We adopt one of the precision configurations proposed by Tsai et al. [20], utilizing double precision at the finest level, single precision at the second level, and half precision for the remaining levels. We test the overall execution time of the 16 representative matrices, with the setup and solve presented in Figure 7 in the bars AmgT (Mixed). Overall, AmgT (Mixed) shows geomean speedups of 1.02× and 1.04× (up to 1.08× and 1.15×) over AmgT (FP64) on the two GPUs, respectively. The performance of mixed-precision SpGEMM and SpMV exceeds that in double precision, with geomean speedups of 1.06× and 1.06× (up to 1.39× and 1.28×) and 1.05× and 1.08× (up to 1.24× and 1.27×) on the two GPUs, respectively. Given our restriction of maximum grid levels to seven, some matrices only generate two or three grid levels, and the size of the matrix is considerably smaller at the coarser levels. With these conditions, the performance improvement derived from mixed precision under our current configuration is convincing enough to illustrate the efficiency of mixed precision.

Taking the matrix 'bcsstk39' as an example, which generates a four-level grid, and thus three precision kernels are

all utilized. This matrix executes nine SpGEMM operations, with our mixed precision SpGEMM, achieves acceleration of 1.13× and 1.28×, compared to the double precision SpGEMM on the two GPUs, respectively. In the solve phase, the matrix performs 851 SpMV operations, our mixed precision SpMV achieves speedups of 1.07× and 1.09× over the double precision SpMV on the two GPUs, respectively.

D. Detailed Performance of the 16 Matrices

To capture the costs of each SpGEMM and SpMV operation in the setup and solve phases more clearly, we record the execution time of all instances of the two kernel calls throughout the process for the 16 matrices using the three approaches. Figure 8 presents these performance results on the H100 GPU. With these records, we can further clarify the reason for the performance behavior of the 16 matrices in Figure 7.

Taking the matrix 'TSOPF_RS_b300_c3' as an example, which generates seven grid levels, executes 18 SpGEMM in the setup phase, and 1701 SpMV in the solve phase. Observing the execution time of SpGEMM for this matrix, AmgT (FP64) is 1.51× faster than HYPRE (FP64), and AmgT (Mixed) is 1.08× faster than AmgT (FP64). The corresponding SpGEMM performance subfigure of this matrix shows that the blue dots (HYPRE) are higher than the yellow (AmgT with double precision) and red (AmgT with mixed precision), and the red dots are slightly higher than the yellow dots for most of the operations after the fourth. Regarding SpMV, AmgT (FP64) is 1.41× faster than HYPRE (FP64), and AmgT (Mixed) is 1.08× faster than AmgT (FP64). Its corresponding SpMV performance subfigure in Figure 8 displays the runtime of the 1701 SpMV operations. The topmost level of dots represents the time to perform SpMV on the finest grid level of the matrix, and it is evident that the blue dots are higher than the red and yellow dots. The predominantly yellow dots at the bottom levels of the subfigure represent the execution times of our half-precision SpMV operations. Compared to the red dots in the higher levels, these indicate that the mixed-precision method significantly reduces the execution time for SpMV.

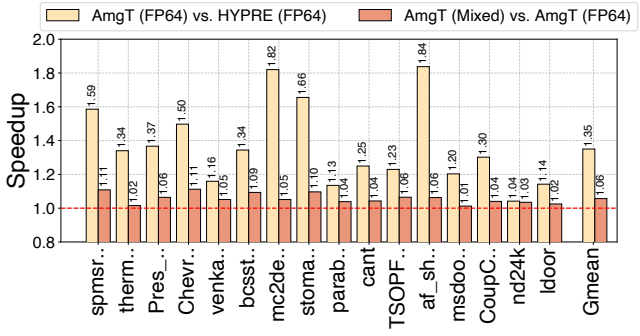

Fig. 9. Performance comparison of HYPRE (FP64), AmgT (FP64) and AmgT (Mixed) on eight A100 GPUs.

E. Performance on Multi-GPU

Because of the incorporation into HYPRE, AmgT naturally supports distributed computing. We thus also compare the performance of HYPRE (FP64), AmgT (FP64) and AmgT (Mixed) on eight A100 GPUs, as shown in Figure 9. Compared to HYPRE (FP64), our AmgT (FP64) method achieves a geomean of 1.35× (up to 1.84×) speedups. Also, our AmgT (Mixed) method gets a geomean of 1.06× (up to 1.11×) speedups over the AmgT (FP64) method. Although data partitioning leads to higher communication costs with less computation per GPU, our algorithms are still able to maintain stable superiority relative to the HYPRE with cuSPARSE.

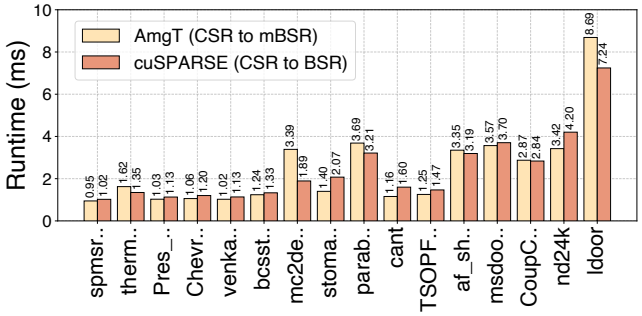

Fig. 10. Format conversion cost comparison of the CSR to mBSR in AmgT and CSR to BSR in cuSPARSE.

F. Performance on AMD GPU

We also deploy our AmgT on AMD GPUs. Since the input sizes supported by the AMD Matrix Core are not suitable for our algorithm, we abandon the use of the Matrix Core in favor of using the standard compute cores exclusively for the computation. We compare our algorithm with HYPRE calling the rocSPARSE kernels, and the subfigure (c) of Figure 7 shows the performance results on an AMD MI210 GPU.

For the setup phase, compared with HYPRE (FP64) calling rocSPARSE kernels, AmgT (FP64) attaches a geomean of 1.78× (up to 2.05×) speedups on the MI210 GPU, where the execution time of SpGEMM operations achieves a geomean of 4.67× (up to 5.96×) speedups. For the solve phase, our AmgT (FP64) gets a geomean of 2.42× (up to 4.53×) speedups on the MI210 GPU over HYPRE, where the execution time of SpMV operations reaches a geomean of 2.92× (up to 6.70×) speedups. Due to the limited support for the FP16 precision in the programming instructions, we did not utilize the FP16 precision in our mixed-precision implementation. Instead, we set the finest level to use FP64 precision, while the remaining levels employ FP32 precision. Additionally, the AMD MI210 GPU has equal compute capabilities for both FP64 and FP32 precisions. As a result, the performances of our AmgT (FP64) and AmgT (Mixed) are nearly identical.

G. Format Conversion Cost

We also compare the costs of converting a matrix from the CSR to the mBSR in AmgT and the BSR in cuSPARSE. Still using the 16 representative matrices, the format conversion costs are plotted in Figure 10. Since the main difference between the mBSR and the BSR formats is the addition of the `BlcMap` array, the two conversion costs are very similar.

The format conversion, as explained in Section IV.E, is called *2×#Levels-1* times in the entire data flow, and generally occupies about longer than 5% overall execution time. Because the tensor core friendly SpGEMM and SpMV in our AmgT both use exactly the same mBSR, the conversion costs are well limited.

VI. Related Work

Fast algorithms for the **key components of AMG** always receive much attention. Coarsening, normally the first stage of the setup phase, has a variety of representative methods, such as the parallel modified independent set (PMIS) [71] the semicoarsening scheme [72] and the algebraic interface [73]. The interpolation stage also has multiple efficient approaches, such as long-range [74], distance-two [75], element [76], optimal implementation [77], and area-specific [78]. Li et al. [35] recently showed that interpolation can use SpGEMM for better performance, and this method is selected in our work. Xu et al. [79] proposed the αSetup-AMG based on an adaptive setup strategy. Smoothing is a key component in the solve phase, and machine learning methods were developed to select good smoothers [80]. Chow et al. [81] surveyed various parallel techniques for AMG.

In addition to the graph-style components mentioned above, **sparse kernels** also play an important role in AMG. Much SpGEMM work focuses on scalability [32], [37], [38], [43], [44], [47], compression [32], [33], [39], vectorization [45], [46], [46], [51] and new accumulators [41], [42], [82]. As for SpMV, cache-friendly data layout [53]–[57], [67], [83], load balancing [58], [59], [62], [65], [66], vectorization [52], [60], [61], [64] and sparse communication [84] received the most attention. In particular, tSparse [36] can use tensor cores for dense parts in SpGEMM and DASP [63] designed a new sparse format in order to efficiently utilize Tensor Core for SpMV. Futhermore, sparse kernels were also accelerated on FPGAs [68] and ReRAM hardware [85]. Compared to the studies, our AmgT runs on a unified sparse format, accelerates both kernels on tensor cores, and shows obviously better performance in AMG.

In addition to the components accelerated on a single node, **AMG on distributed memory systems** requires additional efforts to reduce communication [86], [87] and realize scalability [7], [8], [88], [89], through modeling [90], discretization [91], [92], fault resilience [93], domain decomposition [94], [95], sparsification [96] and task parallelism [97]. Although the AmgT proposed in this paper focuses on optimizations on GPUs, it could be incorporated into existing libraries like HYPRE and benefit large-scale platforms.

A number of **mathematical software for AMG** have been developed. HYPRE [22] with BoomerAMG [98] includes easy-to-use interfaces [99]–[101] and GPU support [17]. It was also integrated into the xSDK [102], prepared for Exascale computing [103], and used as a representative workload to benchmark heterogeneous supercomputers [104]. Bell et al. [16] implemented AMG with fine-grained parallelism on GPUs, and developed the CUSP library [23] with a Python interface [105]. Naumov et al. [15] implemented classical and aggregation-based AMG methods on GPUs in the AmgX library. A multiple precision AMG [20] algorithm was recently developed in the Ginkgo library [25]. Park et al. [7], Yang et al. [14], Yuan et al. [21], Liu et al. [9], Bernaschi et al. [19], Wang et al. [10] and Boukhris et al. [18] also optimized AMG on multicores or GPUs. In addition to general purpose processors, AMG was also accelerated on FPGAs [12] and ReRAM hardware [13]. In comparison, our AmgT is a new library that largely uses tensor cores and their mixed-precision ability for state-of-the-art performance.

Existing research also proposed **scientific kernels on tensor cores**, such as reduction and scan [106], GEMM with extended precision [107], matrix factorization [108], [109], stencil [110], [111], FFT [112], basic linear algebra operations [113], [114], sparse matrix multiplication [36], [63], [115]–[117], iterative refinement [118], and random projection [119]. Compared with those studies, our work shows that more irregular kernels could be accelerated by tensor cores, and using a unified storage format can be more efficient in the complete AMG procedure.

In addition, **mixed precision computations** also show great potential for scientific kernels. A series of linear algebra methods [120], in particular iterative solvers such as conjugate gradient [121]–[123], generalized minimal residual (GMRES) [124]–[126] and preconditioner [127], as well as AMG [20] demonstrated the effectiveness of mixed precision. In our work, SpGEMM and SpMV kernels have higher potential to obtain benefits from mixed precision through the use of tensor cores.

VII. Conclusion

We in this paper have proposed AmgT, a new solver that utilizes the tensor core and mixed precision ability of the latest GPUs for the entire procedure of AMG. Both SpGEMM and SpMV operations with multiple precisions are accelerated on top of a unified sparse format called mBSR and on tensor core units. On single-GPU (A100, H100 and MI210) and multi-GPU (eight A100) platforms, our AmgT significantly outperforms the latest GPU version of the HYPRE library.

Acknowledgements

We greatly appreciate the invaluable comments of all reviewers. Weifeng Liu is the corresponding author of this paper. This work was partially supported by the National Key R&D Program of China (Grant No. 2023YFB3001604), the National Natural Science Foundation of China (Grant No. U23A20301, No. 62372467, No. 62204265 and No. 62234010) and the State Key Laboratory of Computer Architecture (ICT, CAS) (Grant No. CARCHA202115). This work was partially supported by the Spanish Ministry of Science and Innovation MCIN/AEI/10.13039/501100011033 (contracts PID2019-107255GB-C21 and PID2019-105660RBC22) and by the Generalitat de Catalunya (contract 2021-SGR-00865). We are also very grateful to Yuyao Niu and Runzhang Mao for help in the evaluation.

REFERENCES

[1] R. Falgout, "An introduction to algebraic multigrid," *Computing in Science & Engineering*, vol. 8, no. 6, pp. 24–33, 2006.

[2] J. Xu and L. Zikatanov, "Algebraic multigrid methods," *Acta Numerica*, vol. 26, pp. 591–721, 2017.

[3] K. Stüben, "A review of algebraic multigrid," in *Numerical Analysis: Historical Developments in the 20th Century*, 2001, pp. 331–359.

[4] A. Gholami, D. Malhotra, H. Sundar, and G. Biros, "Fft, fmm, or multigrid? a comparative study of state-of-the-art poisson solvers for uniform and nonuniform grids in the unit cube," *SIAM Journal on Scientific Computing*, vol. 38, no. 3, pp. C280–C306, 2016.

[5] T. Mifune, T. Iwashita, and M. Shimasaki, "A fast solver for fem analyses using the parallelized algebraic multigrid method," *IEEE Transactions on Magnetics*, vol. 38, no. 2, pp. 369–372, 2002.

[6] S. Buckeridge and R. Scheichl, "Parallel geometric multigrid for global weather prediction," *Numerical Linear Algebra with Applications*, vol. 17, no. 2-3, pp. 325–342, 2010.

[7] J. Park, M. Smelyanskiy, U. M. Yang, D. Mudigere, and P. Dubey, "High-performance algebraic multigrid solver optimized for multi-core based distributed parallel systems," in *SC '15*, 2015, pp. 1–12.

[8] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of scaling algebraic multigrid across modern multicore architectures," in *IPDPS '11*, 2011, pp. 275–286.

[9] H. Liu, B. Yang, and Z. Chen, "Accelerating algebraic multigrid solvers on nvidia gpus," *Computers & Mathematics with Applications*, vol. 70, no. 5, pp. 1162–1181, 2015.

[10] L. Wang, X. Hu, J. Cohen, and J. Xu, "A parallel auxiliary grid algebraic multigrid method for graphic processing units," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, pp. C263–C283, 2013.

[11] D. Demidov, "Amgcl: An efficient, flexible, and extensible algebraic multigrid implementation," *Lobachevskii Journal of Mathematics*, vol. 40, pp. 535–546, 2019.

[12] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "Fp-amg: Fpga-based acceleration framework for algebraic multigrid solvers," in *FCCM '20*, 2020, pp. 148–156.

[13] M. Fan, X. Tian, Y. He, J. Li, Y. Duan, X. Hu, Y. Wang, Z. Jin, and W. Liu, "Amgr: Algebraic multigrid accelerated on reram," in *DAC '23*, 2023, pp. 1–6.

[14] X. Yang, S. Li, F. Yuan, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid computation and parallelization on multi-cores," in *ICS '23*, 2023, pp. 227–239.

[15] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, "Amgx: A library for gpu accelerated algebraic multigrid and preconditioned iterative methods," *SIAM Journal on Scientific Computing*, vol. 37, no. 5, pp. S602–S626, 2015.

[16] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[17] R. D. Falgout, R. Li, B. Sjögreen, L. Wang, and U. M. Yang, "Porting hypre to heterogeneous computer architectures: Strategies and experiences," *Parallel Computing*, vol. 108, p. 102840, 2021.

[18] S. Boukhris, A. Napov, and Y. Notay, "Algebraic multigrid using a stencil-csr hybrid format on gpus," *SIAM Journal on Scientific Computing*, vol. 45, no. 3, pp. C154–C178, 2023.

[19] M. Bernaschi, A. Celestini, F. Vella, and P. D'Ambra, "A multi-gpu aggregation-based amg preconditioner for iterative linear solvers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, 2023.

[20] Y.-H. M. Tsai, N. Beams, and H. Anzt, "Three-precision algebraic multigrid on gpus," *Future Generation Computer Systems*, vol. 149, pp. 280–293, 2023.

[21] F. Yuan, X. Yang, S. Li, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid preconditioned conjugate gradient method on multi-cores," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 5, pp. 768–779, 2024.

[22] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *ICCS '02*, 2002, pp. 632–641.

[23] S. Dalton, N. Bell, L. Olson, and M. Garland, "Cusp: Generic parallel algorithms for sparse matrix and graph computations," 2014.

[24] X. Xu, X. Yue, R. Mao, Y. Deng, S. Huang, H. Zou, X. Liu, S. Hu, C. Feng, S. Shu, and Z. Mo, "Jxpamg: a parallel algebraic multigrid solver for extreme-scale numerical simulations," *CCF Transactions on High Performance Computing*, vol. 5, no. 1, pp. 72–83, 2023.

[25] H. Anzt, T. Cojean, G. Flegar, F. Göbel, T. Grützmacher, P. Nayak, T. Ribizel, Y. M. Tsai, and E. S. Quintana-Ortí, "Ginkgo: A modern linear operator algebra framework for high performance computing," *ACM Trans. Math. Softw.*, vol. 48, no. 1, 2022.

[26] J. Burgess, "Rtx on – the nvidia turing gpu," in *Hot Chips '19*, 2019, pp. 1–27.

[27] J. Choquette and W. Gandhi, "Nvidia a100 gpu: Performance & innovation for gpu computing," in *Hot Chips '20*, 2020, pp. 1–43.

[28] J. Choquette, "Nvidia hopper gpu: Scaling performance," in *Hot Chips '22*, 2022, pp. 1–46.

[29] A. Biswas, "Sapphire rapids," in *Hot Chips '21*, 2021, pp. 1–22.

[30] H. Jiang, "Intel's ponte vecchio gpu : Architecture, systems & software," in *Hot Chips '22*, 2022, pp. 1–29.

[31] A. Smith and N. James, "Amd instinct mi200 series accelerator and node architectures," in *Hot Chips '22*, 2022, pp. 1–23.

[32] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[33] ——, "Challenges and advances in parallel sparse matrix-matrix multiplication," in *ICPP '08*, 2008, pp. 503–510.

[34] ——, "On the representation and multiplication of hypersparse matrices," in *IPDPS '08*, 2008, pp. 1–11.

[35] R. Li, B. Sjögreen, and U. M. Yang, "A new class of amg interpolation methods based on matrix-matrix multiplications," *SIAM Journal on Scientific Computing*, vol. 43, no. 5, pp. S540–S564, 2021.

[36] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix-matrix multiplication with gpu tensor cores," *Computers & Electrical Engineering*, vol. 88, p. 106848, 2020.

[37] A. Azad, G. Ballard, A. Buluç, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams, "Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication," *SIAM Journal on Scientific Computing*, vol. 38, no. 6, pp. C624–C651, 2016.

[38] G. Ballard, A. Buluç, J. Demmel, L. Grigori, B. Lipshitz, O. Schwartz, and S. Toledo, "Communication optimal parallel multiplication of sparse random matrices," in *SPAA '13*, 2013, pp. 222–231.

[39] M. T. Hussain, O. Selvitopi, A. Buluç, and A. Azad, "Communication-avoiding and memory-constrained sparse matrix-matrix multiplication at extreme scale," in *IPDPS '21*, 2021, pp. 90–100.

[40] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç, "Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors," *Parallel Computing*, vol. 90, p. 102545, 2019.

[41] Y. Nagasaka, A. Nukada, and S. Matsuoka, "High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu," in *ICPP '17*, 2017, pp. 101–110.

[42] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and gpu architectures," *Parallel Computing*, vol. 78, pp. 33–46, 2018.

[43] G. Ballard, C. Siefert, and J. Hu, "Reducing communication costs for sparse matrix multiplication within algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 38, no. 3, pp. C203–C231, 2016.

[44] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz, "Hypergraph partitioning for sparse matrix-matrix multiplication," *ACM Trans. Parallel Comput.*, vol. 3, no. 3, 2016.

[45] Y. Niu, Z. Lu, H. Ji, S. Song, Z. Jin, and W. Liu, "Tilespgemm: a tiled algorithm for parallel sparse general matrix-matrix multiplication on gpus," in *PPoPP '22*, 2022.

[46] V. Le Fèvre and M. Casas, "Efficient execution of spgemm on long vector architectures," in *HPDC '23*, 2023, p. 101–113.

[47] W. Liu and B. Vinter, "An efficient gpu general sparse matrix-matrix multiplication for irregular data," in *IPDPS '14*, 2014, pp. 370–381.

[48] Z. Xie, G. Tan, W. Liu, and N. Sun, "A pattern-based spgemm library for multi-core and many-core architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 1, pp. 159–175, 2022.

[49] ——, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *ICS '19*, 2019, pp. 94–105.

[50] J. Liu, X. He, W. Liu, and G. Tan, "Register-aware optimizations for parallel sparse matrix-matrix multiplication," *International Journal of Parallel Programming*, vol. 47, no. 3, pp. 403–417, 2019.

[51] H. Cheng, W. Li, Y. Lu, and W. Liu, "Haspgemm: Heterogeneity-aware sparse general matrix-matrix multiplication on modern asymmetric multicore processors," in *ICPP '23*, 2023, pp. 807–817.

[52] R. Li and Y. Saad, "Gpu-accelerated preconditioned iterative linear solvers," *The Journal of Supercomputing*, vol. 63, pp. 443–466, 2013.

[53] A. N. Yzelman and R. H. Bisseling, "Cache-oblivious sparse matrix-vector multiplication by using sparse matrix partitioning methods," *SIAM Journal on Scientific Computing*, vol. 31, no. 4, pp. 3128–3154, 2009.

[54] ——, "Two-dimensional cache-oblivious sparse matrix-vector multiplication," *Parallel Computing*, vol. 37, no. 12, pp. 806–819, 2011.

[55] A. N. Yzelman and D. Roose, "High-level strategies for parallel shared-memory sparse matrix-vector multiplication," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 1, pp. 116–125, 2014.

[56] A. Buluç, S. Williams, L. Oliker, and J. Demmel, "Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication," in *IPDPS '11*, 2011, pp. 721–733.

[57] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *SPAA '09*, 2009, pp. 233–244.

[58] J. I. Aliaga, H. Anzt, T. Grützmacher, E. S. Quintana-Ortí, and A. E. Tomás, "Compression and load balancing for efficient sparse matrix-vector product on multicore processors and graphics processing units," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 14, p. e6515, 2022.

[59] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang, "Load-balancing sparse matrix vector product kernels on gpus," *ACM Trans. Parallel Comput.*, vol. 7, no. 1, 2020.

[60] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C401–C423, 2014.

[61] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *ICS '15*, 2015, pp. 339–350.

[62] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC '16*, 2016, pp. 678–689.

[63] Y. Lu and W. Liu, "Dasp: Specific dense matrix multiply-accumulate units accelerated general sparse matrix-vector multiplication," in *SC '23*, 2023, pp. 1–14.

[64] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Efficiently running spmv on long vector architectures," in *PPoPP '21*, 2021, pp. 292–303.

[65] H. Mi, X. Yu, X. Yu, S. Wu, and W. Liu, "Balancing computation and communication in distributed sparse matrix-vector multiplication," in *CCGrid '23*, 2023, pp. 535–544.

[66] W. Li, H. Cheng, Z. Lu, Y. Lu, and W. Liu, "Haspmv: Heterogeneity-aware sparse matrix-vector multiplication on modern asymmetric multicore processors," in *CLUSTER '23*, 2023, pp. 1–12.

[67] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," in *IPDPS '21*, 2021, pp. 68–78.

[68] E. Yi, Y. Duan, Y. Bai, K. Zhao, Z. Jin, and W. Liu, "Cuper: Customized dataflow and perceptual decoding for sparse matrix-vector multiplication on hbm-equipped fpgas," in *DATE '24*, 2024, pp. 1–6.

[69] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, 2011.

[70] X. Fu, B. Zhang, T. Wang, W. Li, Y. Lu, E. Yi, J. Zhao, X. Geng, F. Li, J. Zhang, Z. Jin, and W. Liu, "Pangulu: A scalable regular two-dimensional block-cyclic sparse direct solver on distributed heterogeneous systems," in *SC '23*, 2023, pp. 1–14.

[71] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, no. 4, pp. 1019–1039, 2006.

[72] P. N. Brown, R. D. Falgout, and J. E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1823–1834, 2000.

[73] X. Xu and Z. Mo, "Algebraic interface-based coarsening amg preconditioner for multi-scale sparse matrices with applications to radiation hydrodynamics computation," *Numerical Linear Algebra with Applications*, vol. 24, no. 2, p. e2078, 2017.

[74] U. M. Yang, "On long-range interpolation operators for aggressive coarsening," *Numerical Linear Algebra with Applications*, vol. 17, no. 2-3, pp. 453–472, 2010.

[75] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Numerical Linear Algebra with Applications*, vol. 15, no. 2-3, pp. 115–139, 2008.

[76] M. Brezina, A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, and J. W. Ruge, "Algebraic multigrid based on element interpolation (amge)," *SIAM Journal on Scientific Computing*, vol. 22, no. 5, pp. 1570–1592, 2001.

[77] J. Brannick, F. Cao, K. Kahl, R. D. Falgout, and X. Hu, "Optimal interpolation and compatible relaxation in classical algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. A1473–A1493, 2018.

[78] A. H. Baker, T. V. Kolev, and U. M. Yang, "Improving algebraic multigrid interpolation operators for linear elasticity problems," *Numerical Linear Algebra with Applications*, vol. 17, no. 2-3, pp. 495–517, 2010.

[79] X. Xu, Z. Mo, X. Yue, H. An, and S. Shu, "αsetup-amg: an adaptive-setup-based parallel amg solver for sequence of sparse linear systems," *CCF Transactions on High Performance Computing*, vol. 2, pp. 98–110, 2020.

[80] R. Huang, R. Li, and Y. Xi, "Learning optimal multigrid smoothers via neural networks," *SIAM Journal on Scientific Computing*, vol. 45, no. 3, pp. S199–S225, 2023.

[81] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A survey of parallelization techniques for multigrid solvers," *Parallel Processing for Scientific Computing*, pp. 179–201, 2006.

[82] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez, N. Liber, J. Madsen, J. Miles, D. Poliakoff, A. Powell, S. Rajamanickam, M. Simberg, D. Sunderland, B. Turcksin, and J. Wilke, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2022.

[83] J. D. Trotter, S. Ekmekçibaşı, J. Langguth, T. Torun, E. Düzakın, A. Ilic, and D. Unat, "Bringing order to sparsity: A sparse matrix reordering study on multicore cpus," in *SC '23*, 2023, pp. 1–13.

[84] I. Ismayilov, J. Baydamirli, D. Sağbili, M. Wahib, and D. Unat, "Multi-gpu communication schemes for iterative solvers: When cpus are not in charge," in *ICS '23*, 2023, pp. 192–202.

[85] M. Fan, X. Cheng, D. Yang, Z. Jin, and W. Liu, "Recg: Reram-accelerated sparse conjugate gradient," in *DAC '24*, 2024, pp. 1–6.

[86] P. S. Vassilevski and U. M. Yang, "Reducing communication in algebraic multigrid using additive variants," *Numerical Linear Algebra with Applications*, vol. 21, no. 2, pp. 275–296, 2014.

[87] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the performance of an algebraic multigrid cycle using hybrid mpi/openmp," in *ICPP '12*, 2012, pp. 128–137.

[88] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's multigrid solvers to 100,000 cores," in *High-performance scientific computing: algorithms and applications*, 2012, pp. 261–279.

[89] A. J. Cleary, R. D. Falgout, V. E. Henson, J. E. Jones, T. A. Manteuffel, S. F. McCormick, G. N. Miranda, and J. W. Ruge, "Robustness and scalability of algebraic multigrid," *SIAM Journal on Scientific Computing*, vol. 21, no. 5, pp. 1886–1908, 2000.

[90] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the performance of an algebraic multigrid cycle on hpc platforms," in *ICS '11*, 2011, pp. 172–181.

[91] H. Sundar, G. Stadler, and G. Biros, "Comparison of multigrid algorithms for high-order continuous finite element discretizations," *Numerical Linear Algebra with Applications*, vol. 22, no. 4, pp. 664–680, 2015.

[92] R. S. Sampath, S. S. Adavani, H. Sundar, I. Lashuk, and G. Biros, "Dendro: Parallel algorithms for multigrid and amr methods on 2:1 balanced octrees," in *SC '08*, 2008, pp. 1–12.

[93] M. Casas, B. R. de Supinski, G. Bronevetsky, and M. Schulz, "Fault resilience of the algebraic multi-grid solver," in *ICS '12*, 2012, pp. 91–100.

[94] W. B. Mitchell, R. Strzodka, and R. D. Falgout, "Parallel performance of algebraic multigrid domain decomposition," *Numerical Linear Algebra with Applications*, vol. 28, no. 3, p. e2342, 2021.

[95] G. Haase, M. Kuhn, and S. Reitzinger, "Parallel algebraic multigrid methods on distributed memory computers," *SIAM Journal on Scientific Computing*, vol. 24, no. 2, pp. 410–427, 2002.

[96] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson, and J. B. Schroder, "Reducing parallel communication in algebraic multigrid through spar-

sification," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S332–S357, 2016.

[97] A. AlOnazi, G. S. Markomanolis, and D. Keyes, "Asynchronous task-based parallelization of algebraic multigrid," in *PASC '17*, 2017, pp. 1–11.

[98] V. E. Henson and U. M. Yang, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[99] R. D. Falgout, J. E. Jones, and U. M. Yang, "Conceptual interfaces in hypre," *Future Generation Computer Systems*, vol. 22, no. 1, pp. 239–251, 2006.

[100] ——, "Pursuing scalability for hypre's conceptual interfaces," *ACM Trans. Math. Softw.*, vol. 31, no. 3, pp. 326–350, 2005.

[101] R. D. Falgout and P. S. Vassilevski, "On generalizing the algebraic multigrid framework," *SIAM Journal on Numerical Analysis*, vol. 42, no. 4, pp. 1669–1693, 2004.

[102] R. Bartlett, I. Demeshko, T. Gamblin, G. Hammond, M. A. Heroux, J. Johnson, A. Klinvex, X. Li, L. C. McInnes, J. D. Moulton, D. Osei-Kuffuor, J. Sarich, B. Smith, J. Willenbring, and U. M. Yang, "xsdk foundations: Toward an extreme-scale scientific software development kit," *Supercomputing Frontiers and Innovations*, vol. 4, no. 1, pp. 69–82, 2017.

[103] A. H. Baker, R. D. Falgout, H. Gahvari, T. Gamblin, W. Gropp, T. V. Kolev, K. E. Jordan, M. Schulz, and U. M. Yang, "Preparing algebraic multigrid for exascale," *LLNL Technical Report*, 2012.

[104] I. Karlin, Y. Park, B. R. de Supinski, P. Wang, B. Still, D. Beckingsale, R. Blake, T. Chen, G. Cong, C. Costa, J. Dahm, G. Domeniconi, T. Epperly, A. Fisher, S. Kokkila-Schumacher, S. Langer, H. Le, E. K. Lee, N. Maruyama, X. Que, D. Richards, B. Sjogreen, J. Wong, C. Woodward, U. Yang, X. Zhang, B. Anderson, D. Appelhans, L. Barnes, P. Barnes, S. Bastea, D. Boehme, J. A. Bramwell, J. Brase, J. Brunheroto, B. Chen, C. R. Cooper, T. DeGroot, R. Falgout, T. Gamblin, D. Gardner, J. Glosli, J. Gunnels, M. Katz, T. Kolev, I.-F. W. Kuo, M. P. Legendre, R. Li, P.-H. Lin, S. Lockhart, K. McCandless, C. Misale, J. Moreno, R. Neely, J. Nelson, R. Nimmakayala, K. O'Brien, K. O'Brien, R. Pankajakshan, R. Pearce, S. Peles, P. Regier, S. Rennich, M. Schulz, H. Scott, J. Sexton, K. Shoga, S. Sundram, G. Thomas-Collignon, B. Van Essen, A. Voronin, B. Walkup, L. Wang, C. Ward, H.-F. Wen, D. White, C. Young, C. Zeller, and E. Zywicz, "Preparation and optimization of a diverse workload for a large-scale heterogeneous system," in *SC '19*, 2019, pp. 1–17.

[105] N. Bell, L. N. Olson, and J. Schroder, "Pyamg: Algebraic multigrid solvers in python," *Journal of Open Source Software*, vol. 7, no. 72, p. 4142, 2022.

[106] A. Dakkak, C. Li, J. Xiong, I. Gelado, and W.-m. Hwu, "Accelerating reduction and scan using tensor core units," in *ICS '19*, 2019, pp. 46–57.

[107] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "Egemm-tc: accelerating scientific computing on tensor cores with extended precision," in *PPoPP '21*, 2021, pp. 278–291.

[108] S. Zhang, E. Baharlouei, and P. Wu, "High accuracy matrix computations on neural engines: A study of qr factorization and its applications," in *HPDC '20*, 2020, pp. 17–28.

[109] S. Zhang, R. Shah, H. Ootomo, R. Yokota, and P. Wu, "Fast symmetric eigenvalue decomposition via wy representation on tensor core," in *PPoPP '23*, 2023, pp. 301–312.

[110] X. Liu, Y. Liu, H. Yang, J. Liao, M. Li, Z. Luan, and D. Qian, "Toward accelerated stencil computation by adapting tensor core unit on gpu," in *ICS '22*, 2022, pp. 1–12.

[111] Y. Chen, K. Li, Y. Wang, D. Bai, L. Wang, L. Ma, L. Yuan, Y. Zhang, T. Cao, and M. Yang, "Convstencil: Transform stencil computation to matrix multiplication on tensor cores," in *PPoPP '24*, 2024, pp. 333–347.

[112] L. Pisha and L. Ligowski, "Accelerating non-power-of-2 size fourier transforms with gpu tensor cores," in *IPDPS '21*, 2021, pp. 507–516.

[113] S. Zhang, V. Karihaloo, and P. Wu, "Basic linear algebra operations on tensorcore gpu," in *ScalA '20*, 2020, pp. 44–52.

[114] S. Zhang and P. Wu, "Recursion brings speedup to out-of-core tensorcore-based linear algebra algorithms: A case study of classic gram-schmidt qr factorization," in *ICPP '21*, 2021, pp. 1–11.

[115] H. Wang, W. Yang, R. Hu, R. Ouyang, K. Li, and K. Li, "A novel parallel algorithm for sparse tensor matrix chain multiplication via tcu-acceleration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 8, pp. 2419–2432, 2023.

[116] S. Li, K. Osawa, and T. Hoefler, "Efficient quantized sparse matrix operations on tensor cores," in *SC '22*, 2022, pp. 1–15.

[117] R. Fan, W. Wang, and X. Chu, "Dtc-spmm: Bridging the gap in accelerating general sparse matrix multiplication with tensor cores," in *ASPLOS '24*, 2024, pp. 253–267.

[118] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing gpu tensor cores for fast fp16 arithmetic to speed up mixed-precision iterative refinement solvers," in *SC '18*, 2018, pp. 603–613.

[119] H. Ootomo and R. Yokota, "Mixed-precision random projection for randnla on tensor cores," in *PASC '23*, 2023, pp. 1–11.

[120] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 344–369, 2021.

[121] I. Yamazaki, E. Carson, and B. Kelley, "Mixed precision *s*-step conjugate gradient with residual replacement on gpus," in *IPDPS '22*, 2022, pp. 886–896.

[122] E. Carson, T. Gergelits, and I. Yamazaki, "Mixed precision s-step lanczos and conjugate gradient algorithms," *Numerical Linear Algebra with Applications*, vol. 29, no. 3, p. e2425, 2022.

[123] D. Yang, Y. Zhao, Y. Niu, W. Jia, E. Shao, W. Liu, G. Tan, and Z. Jin, "Mille-feuille: A tile-grained mixed precision single-kernel conjugate gradient solver on gpus," in *SC '24*, 2024.

[124] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, 2018.

[125] E. Oktay and E. Carson, "Multistage mixed precision iterative refinement," *Numerical Linear Algebra with Applications*, vol. 29, no. 4, p. e2434, 2022.

[126] E. Carson and N. Khan, "Mixed precision iterative refinement with sparse approximate inverse preconditioning," *SIAM Journal on Scientific Computing*, vol. 45, no. 3, pp. C131–C153, 2023.

[127] V. Georgiou, C. Boutsikas, P. Drineas, and H. Anzt, "A mixed precision randomized preconditioner for the lsqr solver on gpus," in *ISC '23*, 2023, pp. 164–181.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

This work makes the following contributions:

- C_1 We design a unified sparse matrix format that supports both SpGEMM and SpMV;
- C_2 We propose tensor core and mixed precision friendly SpGEMM and SpMV kernels;
- C_3 We develop the AmgT solver and incorporate it into the Hypre library;
- C_4 We show significant performance gain of AmgT on two latest NVIDIA GPUs.

B. Computational Artifacts

https://zenodo.org/doi/10.5281/zenodo.12581721

II. Artifact Identification

Relation To Contributions

This artifact names AmgT, a new AMG solver that utilizes the tensor core and mixed precision ability of the latest GPUs during multiple phases of the AMG algorithm. (C_1 and C_2) Considering that the sparse general matrix-matrix multiplication (SpGEMM) and sparse matrix-vector multiplication (SpMV) are extensively used in the setup and solve phases, respectively, we propose a novel method based on a unified sparse storage format that leverages tensor cores and their variable precision. (C_3) Our method improves both the performance of GPU kernels, and also minimizes the cost of format conversion in the whole data flow of AMG. To better utilize algorithm components in existing libraries, the data format and compute kernels of the AmgT solver are incorporated into the Hypre library. (C_4) The experimental results on NVIDIA A100 and H100 GPUs show that our AmgT outperforms the original GPU version of Hypre by a factor of on average $1.46\times$ and $1.32\times$ (up to $2.10\times$ and $2.06\times$), respectively.

Expected Results

This artifact contains AmgT in double and mixed precision as well as Hypre calling cuSPARSE kernels. In the test results, the execution time of AmgT (FP64) is less than Hyper, and the execution time of AmgT (Mixed) is less than AmgT (FP64).

Expected Reproduction Time (in Minutes)

Total time: 2 hours. The estimated time to download the datasets, compile this artifact, run the executable files and plot the performance results are 10 min, 10 min, 1.5 hours and 1 min, respectively.

Artifact Setup (incl. Inputs)

- **Hardware**
  GPU: NVIDIA A100 GPU (PCIe, 80GB, 1.94TB/s) or NVIDIA H100 GPU (SXM5, 64GB, 2.02TB/s). Overall, the GPU being used should be one or two NVIDIA GPUs with FP64 Tensor Core.
  CPU: Intel(R) Xeon(R) Platinum 8358P CPU. Please use a multicore CPU.
  Disk Space: at lease 3 GB (to store the experiment input dataset).
- **Software**
  Open MPI v4.0.0 or above;
  NVIDIA CUDA Toolkit v12.2 or above;
  GCC v9.4.0 or above;
  Python 3.9 or above;
  Pandas package v2.2.2.
- **Datasets / Input**
  Our experimental dataset includes 16 matrices in the SuiteSparse Matrix Collection which is publicly available (https://sparse.tamu.edu/about).

Artifact Execution

This artifact consists of four tasks: downloading the test matrices (T_1), compiling the program (T_2), running the program (T_3) and organizing and plotting the result data (T_4). The task T_1 corresponds to the script `matrix.py`, which downloads 16 representative matrices from the SuiteSparse Matrix Collection that will be used as input by the task T_3. The task T_2 is to compile the AmgT and the preprocessing program by executing the script `compile.sh`. After that, we can get all seven executable files which will be used in task T_3. The task T_3 runs all programs in turn to obtain the performance results of each program on these 16 matrices by executing the script `run.sh`. The performance results will be used in the task T_4. The task T_4 extracts the performance data from all the output files obtained from the previous task and plots these results by executing the script `figures.sh`.

The overall execution flow of the artifact is: T_1 (prepare) $\rightarrow$ T_2 (compile) $\rightarrow$ T_3 (run) $\rightarrow$ T_4 (analysis)

Artifact Analysis (incl. Outputs)

After executing the complete workflow, all the performance data has been stored in the form of figures in three files `Fig7.pdf`, `Fig8.pdf` and `Fig9.pdf`.

Artifact Evaluation (AE)

Artifact Setup (incl. Inputs)

- **Preparation for Dataset**
  Download the 16 matrices from the SuitSparse. Run the following command: **$python3 matrix.py**
  The 16 matrices are automatically downloaded to the folder `./matrix/`.
- **Installation and Compilation**
  Modify the configuration information of `comile.sh`.
  Step 1: Modify the CUDA_HOME
  Step 2: Modify the GPU to A100 or H100.
  Run the following command: **$source compile.sh**
  After executing the command, it will generate 7 executable files in `./hypre_test/runnable_files/`, which including:
  `CuSparse`: Test case using the cuSPARSE kernels.
  `No_Mixed`: Test case using the AmgT (FP64 precision).
  `Mixed`: Test case using the AmgT (mixed precision).
  `CuSparse_PrintKernel`: Test case using the cuSPARSE kernels which records the overhead of each kernel call.
  `No_Mixed_PrintKernel`: Test case using the FP64-precision AmgT which records the overhead of each kernel call.
  `Mixed_PrintKernel`: Test case using the mixed-precision AmgT which records the overhead of each kernel call.
  `Preprocess`: Test case about format coversion.

Artifact Execution

- **Evaluation**
  Execute the following command to run all executable files: **$nohup bash run.sh 2>&1**
  The output files will be stored in the path: `./hypre_test/data/`.
  All the test cases are generated in the HYPRE framework, where the version of the main operation calling the cuSPARSE implementation is our baseline. The rest of the test cases are the versions calling the FP64 precision kernel or the mixed precision kernel newly proposed in this work. During the running tests, we mainly record the time of the Setup phase, the execution time of the SpGEMM operation, the time of the Solve phase and the execution time of the SpMV operation during the whole solving process of the Amg solver.
  Since we propose a new data structure different from HYPRE, which incurs additional preprocessing costs, we test and record this overhead as well.

Artifact Analysis (incl. Outputs)

- **Plotting and Analysis**
  Under the path `./figures/`, execute the following command to plot performance: **$bash figures.sh**
  Then, the files `Fig7.pdf`, `Fig8.pdf` and `Fig9.pdf` corresponded to the Figure 7, 8 and 9 in the paper will be generated.

The `Fig7.pdf` shows the performance comparison of the baseline Hypre using FP64 cuSPARSE kernels, our AmgT using FP64, and our AmgT using mixed-precision running on the 16 representative matrices. The expected performance trend should be as shown in the paper that the execution time of AmgT (FP64) is less than Hyper, and the execution time of AmgT (Mixed) is slightly less than AmgT (FP64). The `Fig8.pdf` records the execution time of all instances of the kernel calls througout the process for the 16 matrices using the three approaches, which can obtain more detailed performance. The `Fig9.pdf` shows the format conversion cost comparison of the CSR to mBSR in AmgT and CSR to BSR in cuSPARSE. Because the newly proposed format mBSR differs very little from the BSR format, the two conversion costs are almost very similar.